

8 FRIENDLIER INTERACTION

Everyday OPL programs can use the same graphical interface seen throughout the Psion:

- Menus offer lists of options for you to choose from. You can also select these options with shortcut keys like Ctrl+A, Ctrl+B (Psion-A, Psion-B on the Series 3c) etc.
- Dialogs let a program ask for all kinds of information numbers, filenames, dates and times etc. in one go.
- Screen messages such as 'Busy' are available.
- On the Series 3c, the status window is also available.

Menu keywords begin with an **M**, and dialog keywords with a **D**. In this manual a lower case is used for these letters for example, `mINIT` and `dEDIT` but you can type them using upper or lower case letters.

MENUS

Menus provide a simple way for any reasonably complex OPL program to let you choose from its various options.

To display menus in OPL generally takes three basic steps:

- Use the mINIT command. This prepares OPL for new menus.
- Use the mCARD command (and the mCASC command on the Series 5) to define each menu.
- Use the MENU function to display the menus.

You use the displayed menus like any others on the Psion. Use the arrow keys to move around the menus. Press Enter or an option's shortcut key (or on the Series 5, tap with the pen) to select an option, or press Esc to cancel the menus without making a choice. In either case, the menus are removed, the screen redrawn as it was, and MENU returns a value to indicate the selection made.

Defining the menus

The first argument to mCARD is the name of the menu. This will appear at the top of the menu; the names of all of the menus form a bar across the top of the screen.

From one to eight options on the menu may be defined, each specified by two arguments. The first is the option name, and the second the keycode for a shortcut key. This specifies a key which, when pressed together with the Ctrl key (Psion key on the Series 3c), should select the option. (Your program must still handle shortcut keys which are pressed without using the menu.) It is easiest to specify the shortcut key with % e.g. %a gives the value for a.

If an upper case character is used for the shortcut key keycode, the Shift key must be pressed as well to select the option (on the Series 5, the shortcut key will appear as 'Shift+Ctrl+A' for example). If you supply a keycode for a lower case character, the option is selected only **without** the Shift key pressed. Both upper and lower case keycodes for the same character can be used in the same menu (or set of menus). This feature may be used to increase the total number of shortcut keys available, and is also commonly used for related menu options e.g. %m (%z on the Series 3c) might be used for zooming to a larger font and %M (%Z) for zooming to a smaller font (as in the built-in applications).

For example,

```
mCARD "Comms", "Setup", %s, "Transfer", %t
```

defines a menu with the title Comms. When you move to this menu using 3or4, you'll see it has the two options Setup and Transfer, with shortcut keys Ctrl+S and Ctrl+T (Psion-S and Psion-T on the Series 3c) respectively (and no Shift key required). On the other hand,

```
mCARD "Comms", "Setup", %S, "Transfer", %T
```

would give these options the shortcut keys Shift+Ctrl+S and Shift+Ctrl+T (Shift-Psion-S and Shift-Psion-T on the Series 3c).

The options on a large menu may be divided into logical groups (as seen in many of the menus for the built-in applications) by displaying a line under the final option in a group. To do this, you must pass the negative value corresponding to the shortcut key keycode for the final option in the group. For example, -%A specifies shortcut key Shift+Ctrl+A (Shift-Psion-A on the Series 3c) and displays a grey line under the associated option in the menu.

Each subsequent mCARD defines the next menu to the right. A large OPL application might use mCARD like this:

```
mCARD "File", "New", %n, "Open", %o, "Save", %s
```

```
mCARD "Edit", "Cut", %x, "Copy", %c, "Paste", -%v, "Eval", %e
```

```
mCARD "Search", "First", %f, "Next", %g, "Previous", %p
```

Î Additional Features on the Series 5

On the Series 5, more advanced menu features are available in addition to the more basic features described above. These are as follows:

- menu items without shortcuts
- menu items that are dimmed
- menu items with checkboxes
- menu items with option buttons (sometimes known as radio buttons)
- cascaded menus
- popup menus (see ‘Displaying menus’ below).

It is possible to have menu items without shortcuts, by specifying shortcut values between 1 and 32. The value specified is still returned if the item is selected.

Dimming, checkboxes and option buttons are controlled by adding the following values to the shortcut keycode (the constants are found in Const.oph. See the ‘Calling Procedures’ chapter for details of how to use this file and Appendix E for a listing of it.):

<i>constant name</i>	<i>value</i>	<i>effect</i>
KMenuDimmed%	\$1000	item dimmed
KMenuCheckBox%	\$0800	item has checkbox
KMenuOptionStart%	\$0900	item starts option button list
KMenuOptionMiddle%	\$0A00	item in middle of option button list
KMenuOptionEnd%	\$0B00	item ends option button list
KMenuSymbolOn%	\$2000	symbol on (checkbox/option button)
KMenuSymbolIndeterminate%	\$4000	symbol indeterminate

Dimming a menu item makes it unavailable and on trying to select it, the info print ‘This item is not available’ is automatically displayed. Items with checkboxes have a tick symbol on or off on their left hand side to show whether or not they have been selected. The start, middle and end option buttons are for specifying a group of related items that can be selected exclusively (i.e. if one item is selected then the others are deselected). The number of middle option buttons is variable.

Adding in the KMenuSymbolOn% flag sets the tick on a checkbox or the button on an option button item on. The display of ticks and option buttons is automatically changed appropriately when you select one of these items, but your program needs to maintain the state of any checkbox or option button between displays of the menu.

A single menu card can have more than one set of option buttons and checkboxes, but option buttons in a set should be kept together. For speed, OPL does not check the consistency of these items’ specification.

Cascaded items on which less important menu items can be displayed may also be created. The cascade must be defined before use in a menu card. The following is an example of a ‘Bitmap’ cascade under the File menu of a possible OPL drawing application.

```
mCASC "Bitmap", "Load", %L, "Merge", %M
mCARD "File", "New", %n, "Open", %o, "Save", %s, "Bitmap>", 16, "Exit", %e
```

The trailing > character specifies that a previously defined cascade item is to be used in the menu at this point: it is not displayed in the menu item. A cascade has a filled arrow head displayed along side it in the menu. The cascade title in mCASC is also used only for identification purposes and is not displayed in the cascade itself. This title needs to be identical to the menu item text apart from the >.

For efficiency, OPL doesn't check that a defined cascade has been used in a menu and an unused cascade will simply be ignored.

Shortcut keys used in cascades may be added with the appropriate constant values to enable checkboxes, option buttons and dimming of cascade items.

As is typical for cascade titles, a shortcut value of 16 is used in the example above. This prevents the display or specification of any shortcut key. However, it is possible to define a shortcut key for a cascade title if required, for example to cycle through the options available in a cascade.

Displaying the menus

The MENU function displays the menus defined by mINIT and mCARD, and waits for you to select an option. It returns the shortcut key keycode of the option selected, in the case supplied by you, whether you used Enter or the shortcut key itself (or the pen on the Series 5) to select it. If you supplied a negative shortcut key keycode for an underlined option, it is converted to its positive equivalent.

If you cancel the menus by pressing Esc, MENU returns 0.

I When a set of menus is displayed, the highlight is positioned to the menu and option that the user selected previously (or, if no menus have previously been displayed, to the first option in the first menu).

This works only if your program has only one set of menus. If you have another set of menus, the cursor is **still** set to the position of the menu and option selected in the first set of menus (if that position exists in the new menus).

To avoid this confusion on the Series 3c or to maintain the position of the highlight across menu calls on the Series 5, use `m%=menu(init%)` and set `init%` to zero the first time a set of menus is displayed. The cursor will in this case be positioned to the first option in the first menu. `init%` is set to a value which specifies the menu and option selected, and should be passed to MENU the next time that same set of menus is called. If your program has more than one set of menus, you should have a different `init%` variable for each set of menus.

I Displaying a popup menu

A popup menu which appears at a specified point on the screen, may also be defined and drawn using mPOPUP. **Note that popup menus have only one pane and need not and should not be within the mINIT...MENU structure.** mPOPUP returns the value of the shortcut code in the same way as MENU. For example,

```
mPOPUP(0,0,0,"Continue",%c,"Exit",%e)
```

The first two arguments specify the position of one corner and the third argument specifies which corner this is. This third argument takes values as follows,

	<i>corner</i>
0	top left
1	top right
2	bottom left
3	bottom right

Thus, the example above specifies a popup menu with 0, 0 as its top left-hand corner and with the items 'Continue' and 'Exit', with the shortcuts Ctrl+C and Ctrl+E respectively.

You can add the same values to the shortcut key keycode as those used with mCARD and mCASC to display dimmed items, checkboxes and option buttons. Note, however, that cascades in popup menus are not supported. For example,

```
mPOPUP(0,0,0,"View1",%v OR $2900,"View2",%b OR $A00,"View3",%n OR  
$B00)
```

would display a popup menu with option buttons, with the symbol initially set on on the View1 item (\$2000 is ORed into it as well as \$900).

Problems with menus

You must ensure that you do not use the same shortcut key twice when defining the menus, as OPL does not check for this.

Each menu definition uses some memory, so ‘No system memory’ errors are possible.

Don’t forget to use mINIT before you begin defining the menus.

If the menu titles defined by mCARD are too wide in total to fit on the screen (wider than 40 characters on the Series 5), MENU will raise a ‘Too wide’ error.

Î Shortcut values **must** be alphabetic character codes or numbers between the values of 1 and 32. Any other values will raise an ‘Invalid arguments’ error.

Note also that on the Series 5, a menu is discarded when an item fails to be added successfully. In effect the previous mINIT statement is discarded together with any previous mCARD statements. This avoids the problem of trying to use a badly constructed menu item.

It is therefore incorrect to ignore mCARD errors by having an ONERR label around an mCARD call (see the ‘Error Handling’ chapter for more details). If you do, the menu is discarded and a ‘Structure fault’ will then be raised on using mCARD or MENU without first using mINIT again.

Î When choosing shortcut keys, do not use those such as the number keys which produce different characters when used with the Psion key. Unless you have a good reason not to, stick with a to z and A to Z.

A menu example

This procedure allows you to press the Menu key and see a menu. You might instead be typing a number or some text into the program, or moving around in some way with the arrow keys, and this procedure returns any such keypresses. You could use this procedure instead of a simple GET whenever you want to allow a menu to be shown, and its shortcut keys to work.

Each of the options in the menus have a corresponding procedure named `proc` plus the shortcut key letter so for example, the option with shortcut key Ctrl+N (Psion-N) is handled by the procedure `procn`.

This procedure uses the technique of calling procedures by strings, as described in the ‘Advanced Topics’ chapter:

```
Î PROC kget%:
    LOCAL k%,h$(9),a$(5)
    h$="nosciefgd"                                REM our shortcut keys
    WHILE 1
        k%=GET
        IF k%=$122                                REM Menu key
            mINIT
            mCARD "File","New",%n,"Open",%o,"Save",%s
            mCARD "Edit","Copy",%c,"Insert",-%i,"Eval",%e
            mCARD "Search","First",%f,"Next",%g,"Previous",%d
            k%=MENU
            IF k% AND (LOC(h$,CHR$(k%))<>0) REM MENU CHECK
```

```

        a$="proc"+CHR$(k%)
        @(a$):                                REM procn:, proco:, ...
    ENDIF                                    REM END OF MENU CHECK
ELSEIF KMOD AND $4    REM shortcut key pressed directly?
    k%=k%+$40        REM remove Ctrl modification
    IF LOC(h$,CHR$(k%)) REM DIRECT SHORTCUT KEY CHECK
        a$="proc"+CHR$(k%)
        @(a$):        REM procn:, proco:, ...
    ENDIF            REM END OF DIRECT SHORTCUT KEY CHECK
ELSE                REM some other key
    RETURN k%
ENDIF
ENDWH
ENDP

```

```

Ì PROC kget%:
    LOCAL k%,h$(9),a$(5)
    h$="nosciefgd"        REM our shortcut keys
    WHILE 1
        k%=GET
        IF k%=$122        REM Menu key
            mINIT
            mCARD "File","New",%n,"Open",%o,"Save",%s
            mCARD "Edit","Copy",%c,"Insert",-%i,"Eval",%e
            mCARD "Search","First",%f,"Next",%g,"Previous",%d
            k%=MENU
            IF k% AND (LOC(h$,CHR$(k%))<>0) REM MENU CHECK
                a$="proc"+CHR$(k%)
                @(a$):        REM procn:, proco:, ...
            ENDIF            REM END OF MENU CHECK
        ELSEIF k% AND $200    REM shortcut key pressed directly?
            k%=k%-$200        REM remove Psion key code
            IF LOC(h$,CHR$(k%)) REM DIRECT SHORTCUT KEY CHECK
                a$="proc"+CHR$(k%)
                @(a$):        REM procn:, proco:, ...
            ENDIF            REM END OF DIRECT SHORTCUT KEY CHECK
        ELSE                REM some other key
    
```

```

        RETURN k%
    ENDIF
ENDWH
ENDP

```

procn:, proco:, etc would need to be specified to use this example in practice on any of the machines:

```

PROC procn:
    ...
ENDP

```

```

PROC proco:
    ...
ENDP

```

```

...

```

 Note that this procedure allows you to press a shortcut key with or without the Shift key. So Ctrl+Shift+N would be treated the same as Ctrl+N, similarly on the Series 3c, Shift-Psion-N would be treated the same as Psion-N.

Neither LOC nor the @ operator (for calling procedures by strings) differentiate between upper and lower case. If you have Shifted shortcut keys you will need to compare against two sets of shortcut key lists. For example, with shortcut keys %A, %C, %a and %d, you would have upper/lowercase shortcut key lists like hu\$="AC" and hl\$="ad", and the "MENU CHECK" section becomes:

```

IF k%<=%Z                                REM if upper case shortcut key
    IF LOC(hu$,CHR$(k%))
        a$="procu"+CHR$(k%)
        @(a$):                            REM procua:, procuc:, ...
    ENDIF
ELSE                                        REM else lower case shortcut key
    IF LOC(hl$,CHR$(k%))
        a$="procl"+CHR$(k%)
        @(a$):                            REM procla:, procl:, ...
    ENDIF
ENDIF
ENDIF

```

(This calls procedures procua:, procuc:, procla: and procl:). If a shortcut key was pressed directly you cannot tell from k% whether Shift was used; so make the same change to the "DIRECT SHORTCUT KEY CHECK" section, but use IF KMOD AND 2 instead of IF k%<=%Z.

DIALOGS

In OPL, dialogs are constructed in a similar way to menus:

- Use the dINIT command to prepare OPL for a new dialog. If you give a string argument to dINIT it will be displayed as a title for the dialog. On the Series 5, the title will be in a grey box at the top of the dialog and on the Series 3c it will be separated from the rest of the dialog by a horizontal line.

- Define each line of the dialog, from top to bottom. There are separate commands for each type of item you can use in a dialog for example, dEDIT for editing a string, dDATE for typing in a date, and so on.
- Use the DIALOG function to display the dialog. In general it returns a number indicating the line you were on when you pressed Enter (counting any title line as line 1), or 0 if you pressed Esc.

Use 5 and 6 to move from line to line, and enter the relevant information, as in any other Psion dialog. You can even press Tab to produce vertical lists of options when appropriate.

Each of the commands like dEDIT and dDATE specifies a variable to take the information you type in. If you press Enter to complete the dialog, the information is saved in those variables. The dialog is then removed, and the screen redrawn as it was.

You can press Esc to abandon the dialog without making any changes to the variables.

If you enter information which is not valid for the particular line of the dialog, you will be asked to enter different information.

Here is a simple example. It assumes a global variable `name$` exists:

```
PROC getname:
    dINIT "Who are you?"
    dEDIT name$, "Name:"
    DIALOG
ENDP
```

This procedure displays a dialog with `Who are you?` as its top-line title, and an edit box for typing in your name. If you end by pressing Enter, the name you have typed will be saved in `name$`; if you press Esc, `name$` is not changed.

When the dialog is first displayed, the existing contents of `name$` are used as the string to edit.

Note that the dialog is automatically created with a width suitable for the item(s) you defined, and is centred in the screen.

LINES YOU CAN USE IN DIALOGS

This section describes the various commands that can define a line of a dialog. In all cases:

- `prompt$` is the string which will appear on the left side of the line.
- `var` denotes an argument which **must** be a LOCAL or GLOBAL variable, **because it takes the value you enter**. Single elements of arrays may also be used, but not field variables or procedure parameters. (`var` is just to show you where you must use a suitable variable you don't actually type `var`.)

Where appropriate, this variable provides the initial value shown in the dialog.

Although examples are given using each group of commands, you can mix commands of any type to make up your dialog.

More details of the commands may be found in the 'Alphabetic Listing' chapter.

Strings, secret strings and filenames

```
dEDIT var str$,prompt$,len%
```

defines a string edit box.

`len%` is an optional argument. If supplied, it gives the width of the edit box (allowing for the widest possible character in the font). The string will scroll inside the edit box, if necessary. If `len%` is not supplied, the edit box is made wide enough for the maximum width `str$` could be. (You may wish to set a suitably small `len%` to stop some dialogs being drawn all the way across the screen)

```
dxINPUT var str$,prompt$
```


defines a secret string edit box, such as for a password. A special symbol will be displayed for each character you type, to preserve the secrecy of the string.

```
dFILE var str$,prompt$,f%
```

Î dFILE var str\$,prompt\$,f%,Uid1&,Uid2&,Uid3&

defines a filename editor or selector box. dFILE automatically has ‘Folder’ and ‘Disk’ selectors (only a ‘Disk’ selector on the Series 3c) on the lines below it. f% controls whether you have a file editor or selector in your dialog, and the kind of input allowed. On the Series 5, files selected may also be restricted by UID. See dFILE in the ‘Alphabetic Listing’ for full details of how use dFILE.

Here is an example dialog using these commands:

```
PROC info:
  LOCAL n$(30),pw$(16),f$(255)
  dINIT "Your personal info"
  dEDIT n$,"Name:",15
  dXINPUT pw$,"Password:"
  dFILE f$,"Log file:",0
  RETURN DIALOG
ENDP
```

On the Series 5, you may want to replace the dFILE line with the following:

```
dFILE f$,"Log file,Folder,Disk",0
```

as default prompts for the folder and disk selector boxes are not provided as on the Series 3c.

This procedure returns ‘True’ if Enter was used, indicating that the GLOBAL variables n\$, pw\$ and f\$ have been updated.

Î dEDITMULTI var ptrData&,prompt\$,widthInChars%,noOfLines%,maxLen%

defines a multi-line edit box to go into a dialog. Normally the resulting text would be used in a subsequent dialog, saved to file or printed using the Printer OPX (see the ‘Using OPXs on the Series 5’ chapter). The use of this dialog command is more complicated than the others (see the ‘Alphabetic Listing’ chapter for full details).

Choosing one of a list

```
dCHOICE var choice%,prompt$,list$
```

defines a choice list. list\$ should contain the possible choices, separated by commas for example, “Yes,No”. The choice% variable specifies which choice should initially be shown 1 for the first choice, 2 for the second, and so on.

For example, here is a simple choice dialog:

```
PROC dcheck:
  LOCAL c%
  c%=2 REM default to "View2"
  dINIT "Change View"
  dCHOICE c$,"View:","View1,View2,View3"
  IF DIALOG REM returns 0 if cancelled
    ... REM change view
  ENDIF
```

ENDP

Î On the Series 5, extended choice lists may also be defined by using more than one dCHOICE statement (see the ‘Alphabetic Listing’ chapter for full details of how to do this).

Î dCHECKBOX *chk%*,*prompt\$*

creates a checkbox entry. This is similar to a choice list with two items, except that the list is replaced by a checkbox with the tick either on or off. The state of the checkbox is maintained across calls to the dialog. Initially you should set the live variable *chk%* to 0 to set the tick symbol off and to any other value to set it on. *chk%* is then automatically set to 0 if the box is unchecked or -1 if it is checked when the dialog is closed.

Numbers, dates and times

dLONG *var long* &, *prompt\$*, *min* &, *max* &

and

dFLOAT *var fp*, *prompt\$*, *min*, *max*

define edit boxes for long integers and floating-point numbers respectively. Use dFLOAT to allow fractions, and dLONG to disallow them. *min* (&) and *max* (&) give the minimum and maximum values which are to be allowed. There is no separate command for ordinary integers use dLONG with suitable *min* & and *max* & values.

dDATE *var long* &, *prompt\$*, *min* &, *max* &

and

dTIME *var long* &, *prompt\$*, *type%*, *min* &, *max* &

define edit boxes for dates and times. *min* & and *max* & give the minimum and maximum values which are to be allowed.

For dDATE, *long* &, *min* & and *max* & are specified in “days since 1/1/1900”. The DAYS function is useful for converting to “days since 1/1/1900”.

For dTIME, *long* &, *min* & and *max* & are in “seconds since 00:00”. The DATETOSECS and SECSTODATE functions are useful for converting to and from “seconds since midnight” (they actually use “seconds since 00:00 on 1/1/1970”).

dTIME also has a *type%* argument. This specifies the type of display required:

<i>type%</i>	<i>time display</i>
0	absolute time without seconds
1	absolute time with seconds
2	duration without seconds
3	duration with seconds

Î Two additional types are also available on the Series 5.

4 absolute times in 24 hour clock

8 time not displaying hours

For example, 03:45 is an *absolute time*, while 3 hours 45 minutes is a *duration*.

This procedure creates a dialog, using these commands:

PROC *delivery*:

LOCAL *d* &, *t* &, *num* &, *wt*

d &=DAYS (DAY, MONTH, YEAR)

```

DO
    t&=secs&:
UNTIL t&=secs&:
    num&=1 :wt=10
    dINIT "Delivery"
    dLONG num&,"Boxes",1,1000
    dFLOAT wt,"Weight (kg)",0,10000
    dDATE d&,"Date",d&,DAYS(31,12,1999)
    dTIME t&,"Time",0,0,DATETOSECS(1970,1,1,23,59,59)
    IF DIALOG                                REM returns 0 if cancelled
        ...                                REM rest of code
    ENDIF
ENDP

```

```

PROC secs&:
    RETURN HOUR*INT(3600)+MINUTE*60
ENDP

```

The `secs&:` procedure uses the `HOUR` and `MINUTE` functions, which return the time as kept by the Psion. It is called twice to guard against an incorrect result, in the (albeit rare) case where the time ticks past the hour between calling `HOUR` and calling `MINUTE`.

The `INT` function is used in `secs&:` to force OPL to use long integer arithmetic, avoiding the danger of an 'Overflow' error.

`d&` and `t&` are set up to give the current date and time when the dialog is first displayed. The value in `d&` is also used as the minimum value for `dDATE`, so that in this example you cannot set a date before the current date.

`DATETOSECS` is used to give the number of seconds representing the time 23:59. The first three arguments, 1970, 1 and 1, represent the first day from which `DATETOSECS` begins calculating.

Results from dDATE

`dDATE` returns a value as a number of days. To convert this to a date:

Î use `DAYSTODATE` which converts a number of days since 1/1/1990, to the corresponding date. See the 'Alphabetic Listing' for full details.

Ì

- If you are dealing only with days on or after 1/1/1970, you can subtract 25567 (`DAYS(1,1,1970)`), multiply by 86400 (the number of seconds in a day), and use `SECSTODATE`.
- To handle days before 1/1/1970 as well, you can call the Operating System to perform the conversion. This procedure is passed one parameter, the number of days, and from it sets four global variables `day%`, `month%`, `year%` and `yrdy%`. It calls the Operating System with the OS function:

```

PROC daytodat:(days&)
    LOCAL dyscent&(2),dateent%(4)

```

```

LOCAL flags%,ax%,bx%,cx%,dx%,si%,di%
dyscent&(1)=days&
si%=ADDR(dyscent&()) :di%=ADDR(dateent&())
ax%=$0600 REM TimDaySecondsToDate fn.
flags%=OS($89,ADDR(ax%)) REM TimManager int.
IF flags% AND 1
    RAISE (ax% OR $ff00)
ELSE
    year%=PEEKB(di%)+1900
    month%=PEEKB(UADD(di%,1))+1
    day%=PEEKB(UADD(di%,2))+1
    yrdy%=PEEKW(UADD(di%,6))+1
ENDIF
ENDP

```

If you do use this procedure, be careful to type it exactly as shown here.

Displaying text

dTEXT prompt\$,body\$,type%

defines prompt\$ to be displayed on the left side of the line, and body\$ on the right. **There is no variable associated with dTEXT.** If you use a null string ("") for prompt\$, body\$ is displayed across the whole width of the dialog.

type% is an optional argument. If specified, it controls the alignment of body\$:

type%	effect
0	left align body\$
1	right align body\$
2	centre body\$

Î Note that alignment of body\$ is only supported when prompt\$ is null, with the body being left aligned otherwise.

In addition, you can add any or all of the following three values to type%, for these effects:

\$200	draw a line below this item.
\$400	make the prompt (not the body) selectable.
\$800	make this item a text separator

dTEXT is not just for displaying information. Since DIALOG returns a number indicating the line you were on when you pressed Enter (or 0 if you pressed Esc), you can use dTEXT to offer a choice of options, rather like a menu:

```

PROC select:
    dINIT "Select action"
    dTEXT "Add","",$402
    dTEXT "Copy","",$402

```

```

dTEXT "Review", "", $402
dTEXT "Delete", "", $402
RETURN DIALOG
ENDP

```

In each case `type%` is `$402 ($400+2)`. The `$400` makes each prompt selectable, allowing you to move the cursor on to it. Note that **only** the prompts are selectable: if you try the example given for the Series 3c below on the Series 5, you will see that the items are **not** selectable because the prompt is null. However, the items will be centre aligned in the dialog.

I In addition, you can add any or all of the following three values to `type%`, for these effects:

\$100	use bold text for <code>body\$</code> .
\$200	draw a line below this item.
\$400	make this line selectable. It will also be bulleted if <code>prompt\$</code> is not <code>""</code> .

`dTEXT` is not just for displaying information. Since `DIALOG` returns a number indicating the line you were on when you pressed Enter (or 0 if you pressed Esc), you can use `dTEXT` to offer a choice of options, rather like a menu:

```

PROC select:
    dINIT "Select action"
    dTEXT "", "Add", $402
    dTEXT "", "Copy", $402
    dTEXT "", "Review", $402
    dTEXT "", "Delete", $402
    RETURN DIALOG
ENDP

```

In each case `type%` is `$402 ($400+2)`. The `$400` makes each text string selectable, allowing you to move the cursor on to it, while 2 makes each string centred.

See the ‘Alphabetic Listing’ chapter for full details of `dTEXT`.

Displaying exit keys

Most dialogs are completed by pressing Enter to confirm the information typed, or Esc to cancel the dialog. These keys are not usually displayed as part of the dialog.

However, some Psion dialogs offer you a simple choice, by showing pictures of the keys you can press. A simple “Are you sure?” dialog might, for example, show the two keys ‘Y’ and ‘N’, and indicate the one you press.

If you want to display a message and offer Enter, Esc and/or Space as the exit keys, you can display the entire dialog with the `ALERT` function.

If you want to use other keys, such as Y and N, or display the keys below other dialog items such as `dEDIT`, create the dialog as normal and use the `dBUTTONS` command to define the keys.

`ALERT` and `dBUTTONS` are explained in detail in the ‘Alphabetic listing’ chapter.

OTHER DIALOG INFORMATION

Positioning dialogs

If a dialog overwrites important information on the screen, you can position it with the `dPOSITION` command. Use `dPOSITION` at any time between `dINIT` and `DIALOG`.

`dPOSITION` uses two integer values. The first specifies the horizontal position, and the second, the vertical. `dPOSITION -1, -1` positions to the top left of the screen; `dPOSITION 1, 1` to the bottom right; `dPOSITION 0, 0` to the centre, the usual position for dialogs.

`dPOSITION 1, 0`, for example, positions to the right-hand edge of the screen, and centres the dialog half way up the screen.

Other dialog features

On the Series 5, `dINIT` can take a second optional parameter to specify additional dialog features. This may be any ORed combination of the following constants,:

<i>value</i>	<i>effect</i>
1	buttons positioned on the right rather than at the bottom
2	no title bar (any title in <code>dINIT</code> is ignored)
4	use the full screen
8	don't allow the dialog box to be dragged
16	pack the dialog densely (not buttons though)

Constants for these flags are supplied in `Const.opb`. See the 'Calling Procedures' chapter for details of how to use this file and Appendix E for a listing of it.

It should be noted that dialogs without titles cannot be dragged regardless of the "No drag" setting. Dense packing enables more lines to fit on the screen for larger dialogs.

For example, the following could be used for a large dialog:

```
dINIT "Series 5 Dialog", $15
```

so that the dialog covers the full screen, has buttons (as defined by `dBUTTONS`) on the right and has items densely packed.

Restrictions on dialogs

The following general restrictions apply to all dialogs:

- Only one dialog may be in use at a time.
- A dialog must be initialised (`dINIT`), defined (`dEDIT` etc.) and displayed (`DIALOG`) in the same procedure.

↑

- No error is raised if the dialog is too wide or too long to fit on the screen: it is up to the programmer to ensure the dialog is displayed in a suitable way.

↓

- A dialog may consist of up to nine lines, including any title. Filename editors count as two lines, and exit keys count as three. A 'Too many items' error is raised if this limit is exceeded.
- If the width of any line would make the dialog too wide, a 'Too wide' error is raised when `DIALOG` is called.

SERIES 5 TOOLBAR USAGE

The toolbar on the Series 5 replaces the Series 3a, 3c and Siena status window. All Series 5 OPL programs should use the ROM module `Z:\System\Opl\Toolbar.opo` to create a toolbar window with a title, four buttons and a clock.

The public interface to `Toolbar.opo` is supplied in `Z:\System\Opl\Toolbar.opb` which is reproduced below. The procedures and their usage are then discussed in detail.

TOOLBAR.OPB

```
REM Toolbar.opb version $100
REM Header file for OPL's toolbar
REM (c)Copyright Psion PLC 1997

REM Public procedures
EXTERNAL TBarLink:(appLink$)
EXTERNAL TBarInit:(title$,scrW%,scrH%)
EXTERNAL TBarSetTitle:(name$)
EXTERNAL TBarButt:(shortcut$,pos%,text$,state%,bit&,mask&,flags%)
EXTERNAL TBarOffer%:(winId&,ptrType&,ptrX&,ptrY&)
EXTERNAL TBarLatch:(comp%)
EXTERNAL TBarShow:
EXTERNAL TBarHide:

REM The following are global toolbar variables usable by OPL programs
REM or libraries. Usable after toolbar initialisation:
REM TbWidth%    the pixel width of the toolbar
REM TbVis%      -1 if visible and otherwise 0
REM TbMenuSym%  current Show toolbar menu symbol (ORed with shortcut)

REM Flags for toolbar buttons
CONST KTbFlgCmdOnPtrDown%=$01
CONST KTbFlgLatchStart%=$12  REM start of latchable set
CONST KTbFlgLatchMiddle%=$22 REM middle of latchable set
CONST KTbFlgLatchEnd%=$32    REM end of latchable set
CONST KTbFlgLatched%=$04REM set for current latched item in set

REM End of Toolbar.opb
```

TYPICAL TOOLBAR.OPO USAGE

Typically a program would use `Toolbar.opo` in the following way:

- `LOADM "Z:\System\Opl\Toolbar.opo"` to load the toolbar module. This module must remain loaded as long as you need to use the toolbar.

- ‘Link’ the toolbar-specific globals into the top-level of your program, using TBarLink:
- Initialise the toolbar, creating an invisible toolbar window with title and clock, using TBarInit:
- Add normal or latching toolbar buttons to the toolbar, using TBarButt:
- Make the toolbar visible, using TBarShow:
- Offer all pointer events to the toolbar so that it can call your commands, change the system clock type or display the task list, using TBarOffer%:
- Latch a button down to represent the current view when the view changes, using TBarLatch:
- Change the toolbar title to the current document name, using TBarSetTitle:
- Show and hide the toolbar as appropriate, using TBarShow: and TBarHide:
- Provide *command-handling* procedures to be called by `Toolbar.opo` when a toolbar button is pressed.

TBarLink:

Usage: TBarLink: (appLink\$)

TBarLink: provides all toolbar globals required in your application. It has to be called before TBarInit: and from a higher level procedure in your application than the one in which the globals are used.

appLink\$ is the name of the so-called *continuation procedure* in your main application. TBarLink: calls this procedure, which should then go on to run the rest of your program. This allows the globals declared in TBarLink: to exist until the application exits. appLink\$ must represent a procedure with name and parameters like:

```
PROC appContTBarLink:
    REM Continue after 'linking' toolbar globals
    myAppInit:          REM run rest of program
    ...
ENDP
```

i.e. taking no parameters and with no return-type specification character, so that it can be called using @ (appLink\$) :.

TBarInit:

Usage: TBarInit: (title\$, scrW%, scrH%)

Called at start of application only, this procedure creates the toolbar window, which guarantees that there will be sufficient memory available to display the toolbar at any subsequent time. The toolbar is made invisible when not shown. This procedure also draws all toolbar components except the buttons.

Note that, for speed, TBarInit: turns graphics auto-updating off (using gUPDATE OFF). If automatic updating is required, use gUPDATE ON after TBarInit: returns.

Note also that TBarInit: sets compute mode off (see SETCOMPUTEMODE: in the ‘System OPX’ section in the ‘Using OPXs on the Series 5’ chapter) allowing the program to run at foreground priority when in foreground. By default OPL programs have compute mode on (i.e. they run at background priority even when in foreground).

title\$ is the title shown in the toolbar. You should change this to the name of your current file, using TBarSetTitle:.

scrW% is the full-screen width (gWIDTH at startup).

scrH% is the full-screen height (gHEIGHT at startup).

TBarSetTitle:

Usage: TBarSetTitle: (name\$)

Sets the title in the toolbar.

name\$ is the name of your current file for file-based applications (i.e. applications with the APP...ENDA construct containing FLAGS 1), or the name of your application for non-file applications.

TBarButt:

Usage: TBarButt: (shortcut\$, pos%, text\$, state%, bit&, mask&, flags%)

Adds a button to the previously initialised toolbar.

shortcut\$ is the command shortcut for your application, which is used by Toolbar.opo to perform the command when a toolbar button is selected. On selecting the toolbar button, Toolbar.opo calls your procedure to perform the required command or action. shortcut\$ is case sensitive in the sense that Toolbar.opo calls your procedure named:

- "cmd"+shortcut\$+%: for unshifted, lower-case shortcuts,
- "cmdS"+shortcut\$+%: for shifted, upper-case shortcuts.

For example, if you have the following two commands that also have associated toolbar buttons:

```
mCARD "View", "DoXXX", %x, "DoYYY", %Y REM shortcuts Ctrl+X, Shift+Ctrl+Y
```

you would need to provide command-handling procedures:

- PROC cmdX%:
REM handle shortcut Ctrl+X (lower case shortcut\$="x")
- PROC cmdSY%:
REM handle shortcut Shift+Ctrl+Y (upper case shortcut\$="Y")

and you would create buttons using, e.g.

- TBarButt: ("x", 1, "DoXXX", 0, XIcon&, XMask&, 0)
REM Button for calling cmdX:
- TBarButt: ("Y", 1, "DoYYY", 0, YIcon&, YMask&, 0)
REM Button for calling cmdSY:

pos% is the button position, with pos%=1 for the top button.

text\$ and state% take values as required for gBUTTON.

bit& and mask& are the button's icon bitmap and mask, used in the same way as for gBUTTON.

flags% lets you control how the button is used in two distinct and mutually exclusive ways, as follows:

1. Two *latchable* buttons are often used by the built-in applications to indicate the current view. For an example, see the latchable 'Desk' and 'Sci' buttons in the built-in Calc application. A set of latchable toolbar buttons can be specified in TBarButt: by setting flags% to one of:
KtbFlgLatchStart% for the first button in the latchable set.
KtbFlgLatchMiddle% for any middle buttons (these are optional and not generally used).
KtbFlgLatchEnd% for the last button in the latchable set.
To latch a button down initially to represent the initial setting, OR KtbFlgLatched% with one of the above settings. E.g.
TBarButt: (sh1\$, pos%, txt1\$, st%, bit1&, msk1&, KtbFlgLatchStart%)
TBarButt: (sh2\$, pos%, txt2\$, st%, bit2&, msk2&, KtbFlgLatchEnd% OR
TbFlgLatched%)
will latch down the second button in the set initially.

In the toolbar window, the button with `KTbFlgLatchStart%` set must be above the buttons (if any) with `KTbFlgLatchMiddle%` set, and these in turn must be above the button with `KTbFlgLatchEnd%`.

Only one button in a set is ever latched and pressing another button unlatches the one that was previously set. After pressing and releasing a previously unlatched button in a latchable set, `Toolbar.opo` will, as usual, call your command-handling procedure. When the command has succeeded in changing view, this procedure should set the new state of the button by calling `TBarLatch: (comp%)` where `comp%` is the button number to be latched. This will also unlatch any button that was previously latched. The example below shows how a 'View1' button press, with 'v' as shortcut, should be handled. The other latchable button in this set might be 'View2' with shortcut 'w':

```
PROC cmdV%:
    IF SetView1%:=0                                REM if no error
        TBarLatch: (KView1TbarButton%) REM your CONST KView1TbarButton%
        CurrentView%=1
    ENDIF
ENDP

PROC cmdW%:
    IF SetView2%:=0                                REM if no error
        TBarLatch: (KView2TbarButton%) REM your const KView2TbarButton%
        CurrentView%=2
    ENDIF
ENDP
```

You should call the same command-procedures when the command is performed via a menu or via a keyboard shortcut. This will ensure that the button is latched as required.

2. A setting of `flags%` can also be used to specify that your procedure should be called when the toolbar button is tapped (rather than when the button is released, which is the default). The 'Go to' button in the Program editor works in this way, displaying the popup list of procedures when the button is touched. To implement this using `TBarButt:`, pass `flags%=KTbFlgCmdOnPtrDown%` and provide a procedure named: `"cmdTbDown"+shortcut$+%` which could provide a popup menu, as follows:

```
PROC cmdTbDownC%:
    REM popup next to button, with point specifying
    REM top right corner of popup
    IF mPOPUP(ScrWid%-TbWidth%, 97, KMPopupPosTopRight%, "Cancel",
        0, "Clear", %c)
        cmdC%:                                REM Do the command itself
    ENDIF
ENDP
```

In this case the shortcut is not case-sensitive. Note that when this flag is used, the menu command-procedure is not used directly because a popup is not required when the command is invoked via the menu or via a keyboard shortcut.

TBarOffer%:

Usage: `TBarOffer%: (winId&, ptrType&, ptrX&, ptrY&)`

Offers a pointer event to the toolbar, returning -1 if used and 0 if not used. If not used, the event is available for use by your application.

It is important to call this procedure whenever you receive a pointer event, even when the event is not in the toolbar window, thus enabling `Toolbar.opo` to release the current button, both visually and otherwise.

TBarOffer%: handles:

- the tapping of the clock to change the type between analog and digital system-wide.
- the tapping of the toolbar title to display the task list.
- the selection toolbar buttons, calling your command procedures.
- drawing of the button in the appropriate state.

As usual, the word ‘pointer’ indicates a pen on the Series 5.

`winId&` is the ID of the window that received the pointer event.

`ptrType&` is pointer event type as returned by `GETEVENT32` (pen up, pen down or drag).

`ptrX&, ptrY&` is co-ordinate of pointer event.

TBarLatch:

Usage: `TBarLatch: (button%)`

Latches down a toolbar button, where `button%=1` for the top button in the toolbar. `TBarLatch:` also unlatches any button in the latchable set that was previously latched. See `TBarButt:` for further details on latching buttons.

TBarShow:

Usage: `TBarShow:`

Makes the toolbar visible. The toolbar must exist before calling this procedure. Use `TBarInit:` to create an invisible toolbar with no buttons. Use `TBarButt:` to add buttons.

TBarHide:

Usage: `TBarHide:`

Makes the toolbar invisible.

Public Toolbar globals

The following toolbar globals, provided by `TBarLink:`, may be used in Series 5 OPL applications:

- `TbWidth%` is the pixel width of the toolbar. The rest of the screen width is available for the application.
- `TbVis%` is -1 if visible and otherwise 0
- `TbMenuSym%` is the current ‘Show toolbar’ menu symbol to be added to menu shortcut for View|Show toolbar, e.g.

`mCard "View", "Show toolbar", %t` or `TbMenuSym%`

`TbMenuSym%=(KMenuCheckBox% OR KMenuSymbolOn%)` if the Toolbar is visible and

`TbMenuSym%=KMenuCheckBox%` if invisible. The menu item will therefore be a checkbox item, with the check present or not as appropriate.

GIVING INFORMATION

I Status window temporary and permanent

Pressing Psion-Menu when an OPL program is running will always display a temporary status window. This status window is in front of all the OPL windows, so your program can’t write over it.

Use `STATUSWIN ON` or `STATUSWIN ON, type%` to display a permanent status window. It will be displayed until you use `STATUSWIN OFF`. `type%` specifies the status window type.

I The small status window is displayed for `type%=1` and the large status window either when `type%` is not supplied or when `type%=2`.

Siena There is only one type of status window which will be displayed whatever `type%` you use.

You might use `STATUSWIN ON` when Control-Menu is pressed, for consistency with the rest of the Series 3c.

The status window is displayed on the right-hand side of the screen.

I The rank of the status window

Important: The permanent status window is **behind** all other OPL windows. In order to see it, you **must** use either `FONT` or both `SCREEN` and `gSETWIN`, to reduce the size of the text window and the default graphics window. You should ensure that your program does not create windows over the top of it.

`FONT` automatically resizes these windows to the maximum size excluding any status window. It should be called after creating the status window because the new size of the text and graphics windows depends on the size of the status window. Note that `FONT -$3fff, 0` leaves the current font and style - it just changes the window sizes and clears them.

If you use `SCREEN` and `gSETWIN` instead of `FONT`, you should use the `STATWININFO` keyword (described next) to find out the size of the status window.

I Finding the position and size of a status window

`curtype%=STATWININFO(type%, extent%())` sets the four element array `extent%()` as follows:

`extent%(1)` = pixels from left of screen to status window

`extent%(2)` = pixels from top of screen to status window

`extent%(3)` = status window width in pixels

`extent%(4)` = status window height in pixels for status window `type%`.

`type%=3` specifies the compatibility mode status window and `type%=-1` specifies whichever type of status window is currently shown. Otherwise, use the same values of `type%` as for `STATUSWIN`.

`STATWININFO` returns the type of the **current** status window. The values are as for `type%`, or zero if there is no current status window.

 If `type%=-1` for the current status window and there is none, `STATWININFO` returns consistent information in `extent%()` corresponding to a status window of width zero and full screen height positioned one pixel to the right of the physical screen.

So to set a graphics window to have height `h%` and to use the full screen width up to the current status window (if any), but leaving a one pixel gap between the graphics window and the status window, you could use:

```
STATWININFO(-1, extent%()) :gSETWIN 0,0, extent%(1), h%
```

Alternatively you could simply use `FONT -$3fff, 0` as described under `STATUSWIN` above, which also sets the height to full screen height and sets the text window size to fit inside it.

I What the status window does

The status window always displays the OPL program name, a clock and, by default, an icon. This will be the default OPL icon, unless your program is an *OPI* with its own icon. (*OPIs* are described in the ‘Advanced topics’ chapter.) In addition, the settings selected in the ‘Status window’ menu option of the

System screen are automatically used in OPL status windows. The status window will, therefore, also display all the indicators required, and a digital or analog clock as selected there.

The status window is inaccessible to, and does not affect, the OPL keywords gORDER and gRANK.

You can set or change the name displayed in the status window with SETNAME for example, SETNAME "ABCD" or SETNAME a\$.

Using a list in the status window

Your program may have several distinct modes/views/screens between which you would like the  key to switch. The built-in applications use the  key extensively Agenda uses it to switch to the different views, while Word switches between 'Normal' and 'Outline' view.

The  list is displayed in the status window. It is a list of modes, views or screens which are stepped through as the  key is pressed.

OPL programs can set up a  list. Use

```
DIAMINIT pos%,str1$,str2$,...
```

to initialise the list (This discards any existing list). The list can be initialised before or after a status window is displayed.

str1\$, str2\$ etc. contain the text to be displayed in the status window for each item in the list.

pos% is the initial item on to which the  indicator should be positioned, with pos%=1 specifying the first item. (Any value greater than the number of strings specifies the final item.)

If pos%=0, or if DIAMINIT is used on its own with no arguments, no bar is defined.

If pos%=-1 the list is replaced by the icon instead in the large status window.

If pos%>=1 **you must supply at least this many strings.**

Defining a list uses some memory, so 'No system memory' errors are possible.

DIAMPOS pos% positions the  indicator in a list. You might move the indicator to the next item when the  key is pressed and to the previous item when Shift- is pressed. The  key has keycode value 292 and KMOD returns 2 when the Shift key is pressed.

Positioning outside the range of the items wraps around in the appropriate way if there are three items in the list, DIAMPOS 4 positions to the first.

DIAMPOS 0 causes the  symbol to disappear.

 Use chr\$(4) to display a  key in a menu. If you use it as a shortcut key, a Shift will be added automatically.

Information messages

GIPRINT displays an information message for 2 seconds, in the bottom right corner of the screen. For example, GIPRINT "Not Found" displays Not Found. If a string is too long for the screen, it will be clipped.

You can add an integer argument to control the corner in which the message appears:

<i>value</i>	<i>corner</i>
0	top left
1	bottom left
2	top right
3	bottom right

I Constants for these corner values are supplied in Const.oph. See the 'Calling Procedures' chapter for details of how to use this file and Appendix E for a listing of it.

For example, `GIPRINT "Who?", 0` prints Who? in the top left corner.

Only one message can be shown at a time. You can make the message go away for example, if a key has been pressed with `GIPRINT ""`.

'Busy' messages

Messages which say a program is temporarily busy, or cannot respond for some reason, are by convention shown in the bottom left corner. The `BUSY` command lets you display your own messages of this sort. Use `BUSY OFF` to remove it.

`BUSY "Paused..."`, for example, displays Paused... in the bottom left corner. This remains shown until `BUSY OFF` is used.

You can control the corner used in the same way as for `GIPRINT`.

You can also add a third argument, to specify a delay time (in half seconds) before the message should be shown. Use this to prevent `BUSY` messages from continually appearing very briefly on the screen.

For example, `BUSY "Wait:", 1, 4` will display Wait: in the bottom left corner after a delay of 2 seconds. As soon as your program becomes responsive to the keyboard, it should use `BUSY OFF`. If this occurs within two seconds of the original `BUSY`, no message is seen.

The maximum string length of a `BUSY` message is 80 characters (on the Series 3c, 19 characters) and an 'Invalid argument' error is returned for any value in excess of this.

Only one message can be shown at a time.